Extending Object-Oriented Frameworks with Aspects for Enabling Automatic Support for Domain-Specific Modeling

André L. Santos
Kai Koskimies
Antónia Lopes

DI–FCUL TR–07–22

October 2007

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

Extending Object-Oriented Frameworks with Aspects for Enabling Automatic Support for Domain-Specific Modeling

André L. Santos
Department of Informatics
Faculty of Sciences
University of Lisbon
Portugal

Kai Koskimies
Inst. of Software Systems
Tampere University
of Technology
Finland

Antónia Lopes
Department of Informatics
Faculty of Sciences
University of Lisbon
Portugal

October 2007

Abstract

In the context of *framework*-based development, Domain-Specific Modeling (DSM) is a paradigm that raises the level of abstraction of application engineering. Using a Domain-Specific Modeling Language (DSML), applications are described by domain concepts in a *model* from which the application code is *generated*. This paper presents an approach for automating the construction of a DSM infrastructure for an object-oriented framework, where a DSML and a code generator for building applications are automatically derived. The approach is able to significantly reduce the cost of adopting and evolving a DSM infrastructure. The high degree of automation is possible by enhancing frameworks with an additional layer of *specialization modules*, relying on our previous work on *framework specialization aspects*. The approach was implemented in our ALFAMA tool, and validated by a case study on the Eclipse RCP framework.

1 Introduction

An object-oriented *framework* consists of a set of classes that embodies an abstract design for solutions to a family of related problems [15]. Frameworks are commonly associated with a domain and are a popular means to implement industrial *software product-lines* [4]. A *framework specialization* is an application developed by *instantiating* a certain framework. The activities related to developing a framework are known as *domain engineering*, whereas *application engineering* refers to the development of specializations by instantiating the framework.

Frameworks offer large-scale reuse, and their adoption is widely spread in software development, from the domain of graphical applications to middleware. However, learning how to correctly use a fairly complex framework is a difficult and time-consuming activity [21]. In order to overcome this obstacle in application engineering, domain engineers can adopt a *Domain-Specific Modeling* (DSM) approach. This raises the abstraction level of application development as specializations can be described directly using domain concepts in a *Domain-Specific Modeling Language* (DSML) [7]. The extension of a framework with a DSML is traditionally achieved through the development of a *domain meta-model* and a *code generator*. In this way, an application can be described as an instance of the meta-model and provided as input to the code generator. The code generator performs a *transformation* to obtain application code that instantiates the framework.

DSM approaches claim that it is possible to increase productivity in application engineering activities by up to an order of magnitude [7]. However, these productivity gains imply a significant additional effort

in domain engineering activities, since the domain meta-model and code generator have to be developed and maintained throughout the evolution of the framework. A DSML is the result of several development iterations, and nevertheless, new increments have to be developed when the domain evolves, implying modifications in the framework, meta-model, and code generator. This makes the evolution of the DSM infrastructure challenging.

Developing and maintaining the code generator is the most difficult task. Code generators usually are intellectually demanding, because they are programs that generate other programs. Moreover, a change in the framework may introduce unnoticeable errors in the code that the generator is currently programmed to produce, causing consistency problems.

In this paper, we propose to address these difficulties with an approach that allows the DSML and its code generator to be automatically derived from the framework itself. Comparing to the state-of-the-practice, our approach represents a major strategic difference. Instead of encoding an implicit mapping between a manually defined DSML and the framework in the code generator, we propose the DSML and the transformation to be directly encoded in the framework through a *specialization layer*. The main contribution of this paper is to present an effective technique that supports this encoding in such a way that the meta-model and code generator can be obtained automatically with little extra work.

We capitalize on our previous work [25] that presents a technique based on *aspect-oriented programming* [16] for modularizing framework hot spots that is referred to as *framework specialization aspects*. From the point of view of framework usage, this technique makes the conceptual domain explicit. Applications are developed through the specialization aspects in highly cohesive modules that localize the implementation of concepts, achieving a one-to-one mapping from concepts to implementation modules. The specialization layer relies on specialization aspects, and this is a key issue for making possible to automatically derive the code generator.

Our approach helps to alleviate many problems related to the development and evolution of DSML code generators. Since the code generator and DSML are derived automatically from the framework, these three elements are always aligned and consistent. Framework (and domain) evolution becomes tool-assisted: any framework modification is automatically transferred to the DSML and to the code generator. The task of developing a code generator manually is replaced by the task of creating the aspect-oriented specialization layer, which can be done fairly systematically as we show in this paper.

In general, the approach is applicable to any object-oriented framework that has a reasonably clean mapping between its extension points and domain model. This seems to be the case for most practical frameworks. In addition to the Eclipse RCP framework [17] discussed in this paper, we have experimented applying the approach with JHotDraw [26].

We implemented our approach as a set of Eclipse [9] plug-ins in our ALFAMA tool [24]. The meta-models are defined in EMF (Eclipse Modeling Framework) [10] models. The tool is able to generate an Eclipse plug-in embodying the support for using the DSML, which can be directly executed in Eclipse's runtime workbench. We illustrate ALFAMA's capabilities by presenting a case study on Eclipse Rich Client Platform (RCP) framework — a platform for building stand-alone applications based on Eclipse's dynamic plug-in model and UI facilities.

The paper proceeds as follows. Section 2 presents an overview of our approach. Section 3 addresses the development of the specialization layer. Section 4 evaluates the approach. Section 5 gives details on the code generation and the mechanisms for integration of manual and generated code. Section 6 presents implementation details of the ALFAMA tool. Section 7 describes the case study on Eclipse RCP. Section 8 discusses related work, and Section 9 concludes the paper.

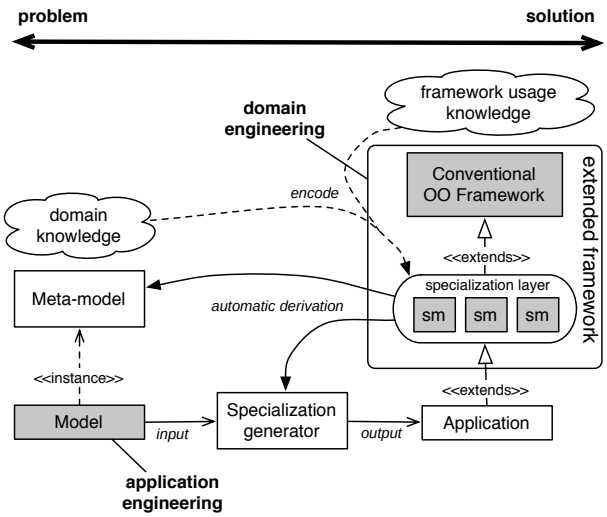


Figure 1: Approach overview.

2 Approach Overview

In this section we provide an overview of the approach that we propose for extending object-oriented frameworks in order to enable automatic DSM support.

Our basic approach is illustrated in Figure 1. In addition to the framework itself, domain engineers have to develop a new layer, which we have called the *specialization layer*. More concretely, the specialization layer consists of a set of specialization modules (*sm* in the figure) defined in terms of annotated classes and aspects. Based on *domain knowledge* and *framework usage knowledge*, domain engineers are expected to develop the specialization modules, which embody (i) the definition of the DSML *meta-model* that can be used to describe the possible instances of the framework, and (ii) the DSML *transformation* for building applications from instances of the meta-model.

The specialization layer is used to automatically derive a *specialization generator* and to extract a dedicated representation of the *meta-model* (e.g. using EMF [10]). Although the specialization layer already defines the DSML meta-model, it is necessary to have an explicit and more appropriate representation, which can be presented graphically, has serialization support, etc. The specialization generator takes instances of the DSML meta-model as input and generates application code that instantiates the framework by extending its specialization modules.

As a case study, we applied this approach to an existing framework: the Eclipse RCP framework [17]. This case study is discussed in detail in Section 7. Through the rest of the paper, for illustration purposes, we shall use just a small fragment of it. For this fragment, the underlying *domain knowledge* is as follows. An *application* has initial window size given by *witdh* and *height*. It may contain several *actions*, an optional *toolbar*, and several *menus*. An action may be user-defined by providing the behavior of an operation. *Exit action* is a framework-provided *action* for quiting the application, which can be used by the application engineers as-is. The *toolbar* may contain application's *actions*. A *menu* has a *name* and may contain *menu actions* which contain references to the application's *actions*. This domain knowledge can be expressed through a domain meta-model as presented in Figure 2.

Having the required domain knowledge to define the DSML meta-model, domain engineers have to combine it with framework usage knowledge in order to develop the specialization modules. For instance,

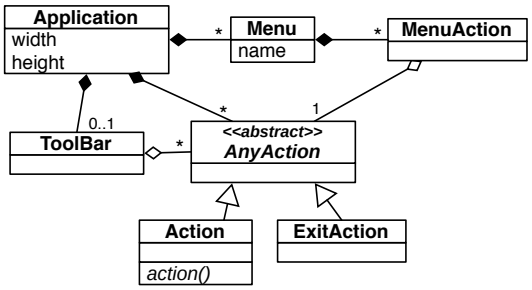


Figure 2: Domain meta-model describing concepts of a fragment of Eclipse RCP framework.

in the RCP fragment the *menus* can be included in an *application* by extending the class Application and overriding the method fillMenuBar(..) in for plugging instances of the class Menu.

In the next section, we show how the domain meta-model and its code generation rules for the RCP fragment can be expressed in terms of specialization modules. For each specialization module, we show an example usage according to the sample meta-model instance presented in Figure 3. In this diagram, an *application* is defined with 400x200 of initial window size, a user-defined *action*, the framework-provided *exit action*, and a *menu* named "My Menu" containing the *exit action*. Recall that the modules that extend the specialization modules are generated by the specialization generator rather than coded manually.

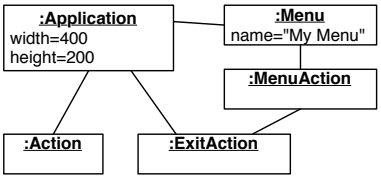


Figure 3: Domain model defining a framework specialization (instance of meta-model presented in Figure 2).

3 Specialization Layer

This section addresses the development of the specialization layer. Subsection 3.1 presents the conceptual model for applying the approach. The remaining subsections address the development of specialization modules reflecting the domain meta-model of Figure 2. Examples of how specialization modules can be used are given according to the domain model of Figure 3. Notice that these usage examples correspond to code that is generated by the specialization generator, and are not meant to be coded manually. Code annotations are used in the specialization modules for defining the meta-model, while the transformations are encoded in AspectJ [8]. The code examples were simplified and detailed issues concerning AspectJ's primitives are explained only briefly. More details on specialization aspects can be found in [25].

3.1 Conceptual model

Figure 4 presents the conceptual model of the specialization layer. The specialization layer is composed by a set of specialization modules. Each specialization module is associated with an application *concept*.

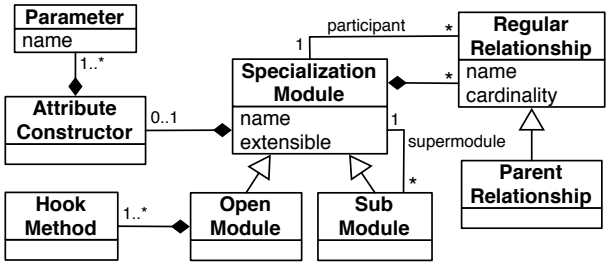


Figure 4: Conceptual model of the specialization layer.

For instance, when encoding a meta-model as the one presented in Figure 2, the specialization layer would contain a specialization module for representing each of the meta-classes (e.g. *application*, *menu*, *action*, etc). A specialization module can be either an abstract class or an abstract aspect. Elements that are meant to be instantiated independently, like for instance the *application* main class, are encoded in classes. On the other hand, elements that are dependent on others, like for instance a *menu action*, are encoded in aspects. A specialization module may have a *attribute constructor* which is a constructor containing only parameters with a primitive type. Each of these parameters represents an attribute of the meta-class represented by the module. A module can be *extensible* or not.

Besides regular modules, there are two other kinds. An *open module* represents an abstract concept that has user-definable operations, which are represented in *hook methods* (i.e. abstract methods intended to be overridden). For example, the *action* would be represented in an open module with a hook method action(). An open module can be seen as a mechanism for realizing of an *open variation point* [13] where a variant can be introduced by implementing the hook methods.

A *submodule* is an extension of an extensible module (its *supermodule*). For example, the *exit action* is represented in a submodule of *action*, since *exit action* is a concrete case of *action* that implements the hook method action(). However, notice that in a meta-model as in Figure 2, the first does not extend the latter, so that the operation is not inherited.

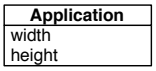
A concept represented in a module may have *relationships* with another concept. A relationship has a *cardinality constraint* defining how many times it may occur, for instance, an *application* may have none or a single *toolbar*. A *parent relationship* is special kind of relationship between a child and a parent concept, for instance, a *menu* (child) of an *application* (parent) — the first cannot exist without the latter. On the other hand, a *regular relationship* establishes a collaboration between two independent concepts, for instance, the *toolbar* may use an *action*.

The next subsections present examples of specialization modules. For each specialization module, an illustration of the meta-model fragment that the module encodes is presented below. Together with each specialization modules, an example extension is also presented. These are illustrated with a fragment of the meta-model instance that would produces the code (through the specialization generator).

3.2 Modules and Attributes

Each specialization module is associated with a single application concept, and we assume the module name to be the concept name. A specialization module is declared using the annotation @Module. If not indicated explicitly, a module is not extensible. The concept's attributes can be declared by annotating a constructor of the module with the annotation @Attributes. Below we present an example of the specialization module associated with *application*.

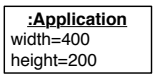
```
@Module
abstract class Application implements IApplication {
 @Attributes
 Application(int width, int height) {
  //...
 }

 final void fillMenuBar(IMenuManager menuBar) { }

 final void makeActions(BarAdvisor barAdvisor) { }

 final void fillToolBar(IToolBarManager coolBar) { }
 //...
}
```



The methods are intended to be empty and non-overridable, since they are going to be advised by other specialization modules. As the names suggest, their role is to allow the customization of *menus*, *actions*, and *toolbar*, respectively.

This specialization module could be used like shown in the example below.

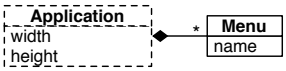
```
class MyApplication extends Application {
  MyApplication() {
    super(400, 200);
  }
}
```



3.3 Parent Relationships

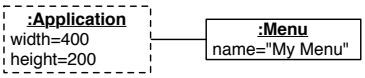
Modules may embody parent relationships with other modules. A parent relationship is defined by annotating an abstract pointcut with @Parent, with the parameters type and card for defining the parent module and the relationship cardinality, respectively. Below we present an example of a specialization module that addresses the inclusion of a *menu* in an *application*.

```
@Module
abstract aspect Menu {
 @Attributes
 Menu(String name) {
  //...
 }

 @Parent(type="Application",card="*")
 abstract pointcut application();

 after(IMenuManager menuBar) : args(menuBar) &&
  within(Application+) && application() &&
  execution(void fillMenuBar(IMenuManager)) {
   menuBar.add(createMenu());
  }
 //...
}
```



The *menu* has a parent *application* that should be given by defining the pointcut application() on an extension of Application (previous specialization module). The cardinality defines that each *application* can have several *menus*. The advice introduces the necessary behavior for plugging the *menu* in the *application* defined by the pointcut application(). Below we present an example usage of this specialization module.

```
aspect MyMenu extends Menu {
 MyMenu() {
  super("My Menu");
 }
 pointcut application() : target(MyApplication);
}
```

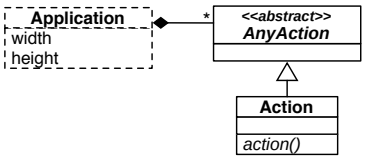


The *menu name* is defined, and the *application* where to plug the *menu* is defined on MyApplication (previous example).

3.4 Open modules

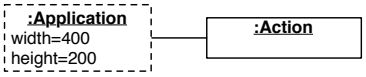
An open specialization module is a module that contains one or more hook methods for its extensions to implement. Hook methods can be declared by annotating an abstract method with @HookMethod. Below we present an example of a specialization module for including an *action* in an *application*.

```
@OpenModule(extensible=true)
abstract aspect Action {
 @HookMethod
 abstract void action();

 @Parent(type="Application",card="*")
 abstract pointcut application();

 after(BarAdvisor barAdvisor) : args(barAdvisor) &&
  within(Application+) && application() &&
  execution(void makeActions(BarAdvisor)) {
   IAction action = createAction();
   barAdvisor.register(action);
  }
 //...
}
```



The module is extensible, as indicated by the parameter extensible in the annotation @Module. A parent relationship is present, analogously to the previous example. The hook method is intended to be implemented by the extensions, in addition to the definition of the pointcut application(). When a module is extensible, it is implied that an additional abstract meta-class is being encoded in the meta-model definition. The meta-classes encoded by the extensible module and its submodules will extend the additional abstract meta-class. Below we present an example usage of this specialization module.

```
aspect MyAction extends Action {
 void action() {
  // to complete manually
 }

 pointcut application() : target(MyApplication);
}
```

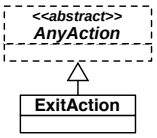


The method action() is implemented and the pointcut application() is defined on MyApplication (introduced earlier).

3.5 Submodules

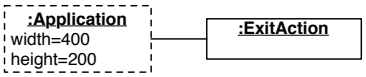
A submodule is an extension of an extensible module (the supermodule). Below we present a specialization module for including the *exit action* in an *application*.

```
@SubModule
abstract aspect ExitAction extends Action {
 void action() { }

 IAction createAction() {
  return ActionFactory.QUIT.create();
 }
}
```



The specialization module extends Action (previous specialization module). The method action() is empty since there is no need for extensions to provide the *action* behavior, and createAction() is overridden for returning the framework's *exit action*. The abstract pointcut application() inherited from Action is still to be defined by the extensions. Below we present an example usage.

```
aspect TheExitAction extends ExitAction {
   pointcut application() : target(MyApplication);
}
```

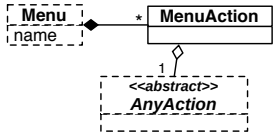


The pointcut application() is defined on MyApplication (introduced earlier) to plug the *exit action*.

3.6 Regular Relationships

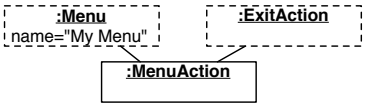
A regular relationship establishes a collaboration between two modules. A module may declare a relationship with another module by annotating an abstract pointcut with @Relationship, with parameters type and card, as in parent relationships. Below we present a specialization module for including a *menu action*, which has a parent relationship to *menu* and a regular relationship with *action*.

```
@Module
abstract aspect MenuAction {
 private IAction action;

 @Relationship(type="Action",card="1")
 abstract pointcut action();

 after() returning(IAction a):
  within(Action+) && action() &&
  execution(IAction createAction()) {
   action = a;
  }

 @Parent(type="Menu",card="+")
 abstract pointcut menu();

 after(Menu m) :
  within(Menu+) && menu() && this(m) &&
  execution(MenuManager createMenu()) {
   m.add(action);
  }
}
```



A single extension of Action is to be defined in the pointcut action(). The first advice captures the *action* creation and keeps its reference. An extension of Menu is to be defined in the pointcut menu(), in order to determine to which *menu* the *menu action* belongs. The second advice adds the *action* upon the creation of the *menu*. Below we present an example usage.

```
aspect MyMenuAction extends MenuAction {
  pointcut action() : target(TheExitAction);
  pointcut menu() : target(MyMenu);
}
```



The pointcut action() is defined on TheExitAction (previous example), which is an indirect extension of Action. The pointcut menu() is defined on MyMenu (introduced earlier).

4 Evaluation

In this section we present an evaluation of our approach. Comparing with the conventional way of implementing DSM support for frameworks, our approach has the following advantages.

Reduces complexity and improves understandability. A code generator is a program that generates another program. In non-trivial cases, this "indirection" is a source of complexity that may cause a burden for domain engineers. The most structured and intuitive approach to the problem is to use code templates. Still, the implementation of the code generator can easily become quite complex. This happens, for instance, when the generated code that results from different model elements has to overlap common modules, or when there is the need to share instance variables. Moreover, we are not aware of any specific methodology for approaching this problem, and most solutions are relatively ad-hoc. In our approach, the DSML transformations encoded

in the specialization layer are expressed directly through a relatively small set of mechanisms based on the use of advices for hook method completion and capturing object instantiations. As illustrated by the examples of Section 3, the same aspect-oriented mechanisms are used repeatedly. The mechanism of submodules allows increments in the DSML to be added without difficulty and at a very low cost. Consider for instance the submodules of Action (Subsections 3.4 and 3.5). One could add another *action* just by coding a simple extension that reuses the supermodule's transformation. After adding this specialization module to the framework, the new feature becomes ready to be used in the DSML. For performing such evolution of the framework, we only need to master a quite small portion of the specialization layer.

Ensures consistency. When using conventional approaches, the consistency between the framework, the modeling language, and the code generator, can be easily broken. A code generator produces text, which is code that instantiates the framework. This code is not checked against compilation until the generator is tested with sample inputs. This brings consistency problems, since a change in the framework may introduce unnoticeable errors in the code that is produced by a not up-to-date generator. Consider the hook method fillMenuBar(..) of Application presented in Subsection 3.2. If, for instance, this method changes its signature, a code generator programmed for overriding the former version of the hook method would not manifest its inconsistency w.r.t. framework until it is executed with a model that triggers the generation of code that overrides that hook method. More concretely, we would only notice the error, during the compilation of the generated code. In contrast, in our approach, if a specialization module specifies an advice that is acting over a non-existent method, one gets a compile-time warning, informing that the module is broken. Obviously, compile-time errors also occur if the body of an advice is using inexistent framework elements.

Promotes composability and contributes to low change impact. In general, code generators are not implemented in cohesive and composable modules. This implies that adding increments to the generator involves modifications in existing generator modules. For instance, recall the example given in Section 3, and suppose that there is no support in the DSML for including *actions* in the *toolbar*. In the case of having a conventional code generator, the support for generating code for including the *actions* would require modifications in the generator part that handles the instances of the meta-class Application. Namely, code that checks if there is a composite instance of the meta-class ToolBar, in order to generate or not the body of the hook method fillToolBar(..). In our approach, a specialization module analogous to the one of Subsection 3.3 could encapsulate the inclusion of *actions* in the *toolbar*, consisting of a non-invasive increment to the specialization layer. Moreover, specialization modules can be composed to form different variants of the DSML. For instance, one can make combinations of specialization modules and obtain different DSMLs without needing to understand any details of the specialization layer.

The disadvantages of our approach are relative to the development of the specialization layer. Specially without a supporting methodology, a domain engineer may take some time to master the development specialization modules, due to their different and apparently complex design style. Another problem is related to the fact that the mechanisms to represent the meta-model elements in the specialization layer are not very flexible. For instance, meta-classes must have a corresponding specialization module and meta-class attributes must have a corresponding contructor parameter.

5 Specialization Support

This section describes the specialization generator that can be automatically derived from a framework extended with a specialization layer (Subsection 5.1). The process of derivation itself is relatively straightforward without any special challenge and, hence, is not described in this paper. However, as explained in Section 6, we have implemented this process in our ALFAMA tool. The rest of the section is devoted to the presentation of the mechanisms available in our approach for integrating manual with generated code (Subsection 5.2).

5.1 Specialization generator

As explained in Section 2, a specialization layer encodes a DSML meta-model. The specialization generator that is derived from the specialization layer takes instances of this meta-model as input, and generates the code of modules (i.e. classes and aspects) that are extensions of the specialization modules.

The task of the specialization generator is conceptually simple, consisting in the following steps:

- For each class of the meta-model instance, it generates a concrete module (either class or aspect) that extends the associated specialization module:
  - It generates a constructor with a call to the superconstructor with the values of the class attributes as arguments.
  - In case of open modules, it introduces an additional intermediary abstract module, which extends the specialization module and includes the pointcut definitions and the constructor call. Application engineers deal with an extension of the intermediary module that contains only the implementation of the hook methods (details are given in Subsection 5.2).
- For each composite reference (parent-child), it generates a pointcut definition on the child module pointing at the parent module.
- For each normal reference (relationship participant), it generates a pointcut definition on the owner module pointing at the participant module.
- A reference with cardinality bigger than one can be seen as a list (e.g. in EMF). In these cases, *aspect precedences* are generated to reflect the composition order of the elements contained in the list.

In the running example, the specialization generator derived from the specialization modules presented throughout Section 3 takes instances of the meta-model presented in Figure 2 as input. The class MyApplication and the aspects MyMenu and MyMenuAction presented in Section 3 are examples of modules generated by the specialization generator when it receives the model of Figure 3 as input. However, there is an exception: the module generated for the open module Action is not the aspect MyAction presented before. The special case of open modules is discussed in the next subsection.

5.2 Integration of manual and generated code

Our approach contemplates two mechanisms for integrating manual and generated code: *escapes* and *accesses*.

5.2.1 Escapes

An escape is a specification-language construct which lets something to be expressed in the underlying implementation language [5]. In our case, these are intended for the code generation that results from open modules (e.g. the Action). The idea is to isolate the hook methods (e.g. action() of Action) from the code that can be generated. In our example, the specialization generator would generate the following intermediary module.

```
abstract aspect MyAction_Adaptor extends Action {
 pointcut application() : target(MyApplication);
}
```

This allows to expose only the hook methods to the application engineers, which can complete the hook methods in a generated skeleton module, as shown below.

```
aspect MyAction extends MyAction_Adaptor {
 void action() {
  // to complete manually
 }
}
```

We believe this is an elegant mechanism for realizing escapes since, in this way, the "pure" application-specific issues concerning a certain concept become completely isolated in a dedicated module, while its composition and parameterization are encapsulated.

5.2.2 Accesses

When writing the code of an open module it is likely that application developers need to access object instances of the application that are initialized within the generated code. An *access* is a mechanism to allow an open module to gain access to object instances that are "hidden" in the generated code. In order to use accesses, domain engineers can explicitly declare at most one object instance to be accessible in a specialization module. This can be done by annotating a method with @Accessible. This declaration expresses that the returned object is the accessible object associated with this module. As an example, consider the specialization module presented below. This module gives support for the inclusion of a *table viewer* in an *application*. Given that the factory method creates the table, it is annotated with @Accessible for the table object to be accessible.

```
@Module
abstract aspect Table {
 //...

 @Accessible
 TableViewer createTable() {
  return new TableViewer();
 }
}
```

Application engineers can access objects by specifying at the modeling level that an open concept accesses another concept. Every meta-class instance that is associated with an open module has by default an association for referencing meta-class instances which have an accessible object. For each meta-class instance defined in this association, the generated skeleton module will have a static variable that will point at the corresponding accessible object automatically. The specialization generator produces an additional aspect for dealing with the accessible objects. For instance, supposing that MyAction presented before accesses a *table viewer*, the generated skeleton module would be as shown below.

```
aspect MyAction extends MyAction_Adaptor {

 // automatic
 static TableViewer tableViewer;

 void action() {
  tableviewer.add("my entry")
 }
}
```

6 Implementation

We implemented our approach in a prototype that we refer to as ALFAMA[1] [24]. The prototype was implemented as a set of Eclipse [9] plugins and is available online, either for downloading or having a deeper insight by watching videos which demonstrate and explain its usage.

The development of the layer of specialization modules relies on a small subset of AspectJ's primitives, and Java 5 annotations. The DSMLs are inferred from the specialization layer and are defined in EMF [10] models. Optionally, GMF [11] can be used independently for developing a concrete syntax for the DSML. As a side observation, we note that AspectJ was used in a light-weight manner in the implementation of the generated plugin.

The ALFAMA tool is composed by three plugins:

(i) *Specialization modules* — A meta-model (described in EMF) for representing the modularization as described in Figure 4. This meta-model is an intermediate representation which abstracts code-specific details.

(ii) *Domain engineering* — A subcomponent for extracting instances of (i) from the specialization layer; a subcomponent for inferring the DSML meta-model and deriving the corresponding specialization generator from an instance of (i); a subcomponent to generate an executable Eclipse plugin that provides support for using the DSML. The specialization generator and some plugin facilities adapt a framework that is part of (iii).

(iii) *Application engineering* — A small meta-model which is extended by all the DSML meta-models, which enables the use of a light-weight framework embodying an abstract generator and the common facilities of the generated plugins, such as the actions for generating the code.

An illustration of the ALFAMA tool is given in the next section, together with the case study description.

7 Case Study

The proposed strategy for modularization went through an iterative process where its applicability was checked against two frameworks, JHotDraw [26] and Eclipse RCP [17]. This section focuses on the latter, which is definitely more complex and can be considered an industrial-strength framework. Eclipse RCP is a framework for building stand-alone applications based on Eclipse's dynamic plug-in model and UI facilities, such as menus, action bars, listeners, tree views, table views and controls (e.g. buttons, labels, etc).

Figure 5 illustrates the ALFAMA development environment. An Eclipse workbench window is shown in (a), where domain engineers develop the specialization modules and generate the DSM infrastructure. We can see the package implementing the specialization layer modules from which an Eclipse plugin embodying the DSM infrastructure is generated. The editor pane shows DSML meta-model inferred from the specialization layer (RCPApplication.ecore). In (b), we can see an Eclipse instance running the plugin generated in (a), where an instance of the meta-model is being edited in Eclipse's default EMF tree view editor. The icons are shown in the editor due to a light-weight mechanism for concrete syntax, which consists of associating an icon with each specialization module. The meta-model instance is used to generate all the application code except the open modules, which are placed in a dedicated package. In the example, the open modules are application's *actions*. We can see in the attribute sheet of AddAction the declaration of two accesses for the *table viewer* and *textbox*. Finally, (c) shows the window of the RCP application generated from the meta-model instance that is shown in (b).

[1] ALFAMA Prototype: Automatic derivation of Languages for Framework specialization by combining Aspect-oriented and Meta-modeling Approaches. Available at http://alfama.sourceforge.net.

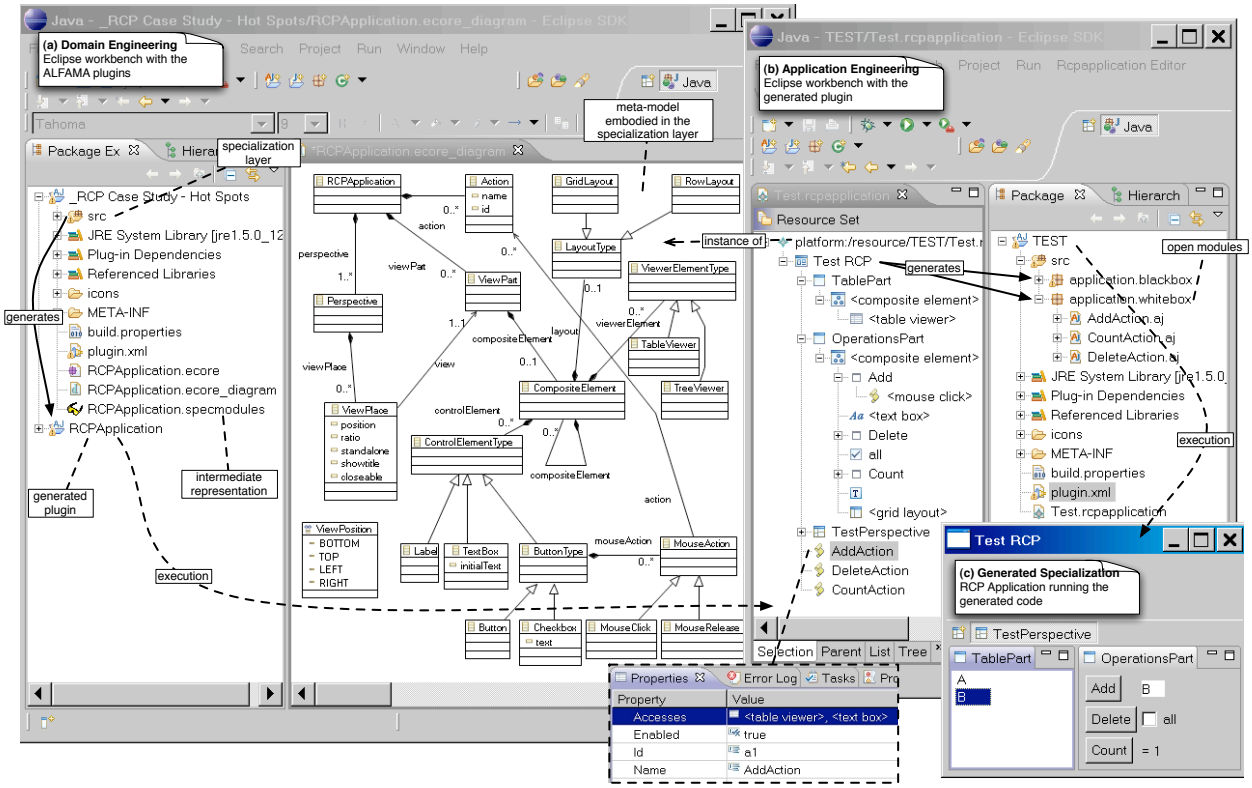


Figure 5: ALFAMA tool: (a) domain engineering and (b, c) application engineering.

Programming in AspectJ is effectively programming in Java plus aspects. In order to give an idea of the development effort that is necessary for developing the specialization layer, Table 1 shows the number of lines of code (LOC) of Java and AspectJ for each specialization module associated with an Eclipse RCP concept, covering those that are visible in the meta-model of Figure 5 (a). Submodules are represented nested under their supermodule. Although we only present these concepts here, the case study was far more extensive and is available on the tool website. The data presented in Table 1 suggests that the effort of developing the specialization modules is not high, specially when compared with the effort of developing a framework-specific specialization generator from scratch. Moreover, notice that only about one fifth of the code uses to AspectJ primitives, while the rest is regular Java. Notice also that the addition of certain modules (e.g. submodules of ViewerElement) do not require any AspectJ code, and also that new modules addressing submodules can be developed at very low cost, i.e. with few lines of code. Based on this data and the kind of specialization modules presented throughout Section 3, we argue that the necessary skills in terms of Java and AspectJ are not very demanding.

8 Related Work

In the context of generative programming, the work in [6] characterizes the expressive power of DSLs (or DSMLs) as a spectrum ranging from *routine configuration* to *creative construction*. *Wizards* are considered as routine configuration with low expressive power, *feature models* are considered to have a mid-range expressive

Table 1: Development effort for Eclipse RCP.

Concept	**Java**	**AspectJ**	**Total LOC**
RCP Application	118	-	118
Action	22	7	29
View Part	23	14	37
Perspective	19	8	27
View Place	35	16	51
Composite Element	18	12	30
Layout	7	7	14
Row Layout	7	-	7
Grid Layout	14	-	14
Control Element	6	8	14
Label	19	-	19
Text Box	19	-	19
Button	39	-	39
Checkbox	14	-	14
Mouse Action	13	12	25
Mouse Click	4	1	5
Mouse Release	4	1	5
Viewer Element	6	8	14
Table Viewer	10	-	10
Tree Viewer	9	-	9
All	406	94	500

power, while *graph-like languages* are considered to have the highest expressive power as a DSL. Using this spectrum of DSL expressiveness, the remainder of this section relates existing approaches, which are summarized in Figure 6.

JavaFrames [14] and SmartBooks [22] are two examples of approaches which allow routine configuration since they provide wizard-like framework specialization. The first relies on the definition of specialization patterns, in order to assist an application engineer via a task-based specialization process with support for partial code generation. The latter consists of a cookbook-like approach with support for active guidance of the application engineer based on high-level activities.

A feature model represents a certain configuration space. Pure::variants [23] is an example of a commercial tool for managing feature variability, where feature models are developed and mapped to components, and applications can be otained by instantiating feature models. Approaches based on *feature-oriented programming* (FOP), such as AHEAD [3], CaesarJ [20], or aspectual-mixin layers (AML) [2], propose systems to be constructed using high-cohesive feature modules, enabling different systems to be obtained by defining

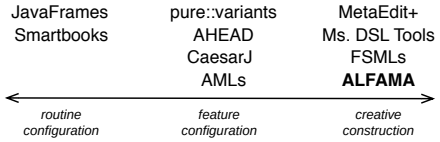


Figure 6: ALFAMA and existing approaches.

a feature configuration. FOP approaches are normally not associated with DSLs, but since their variability space can have a straightforward mapping to a feature model, we consider them in the feature configuration category. Notice that the variants of a system built using FOP are a finite set of applications within the configuration space. Frameworks usually support the development of an infinite set of applications, both by composing existing framework components or introducing application-specific ones.

Existing aproaches for having DSMLs that allow creative construction are based on meta-modeling. MetaEdit+ [18] and Microsoft DSL Tools [12] are two examples of commercial tools that support the development of DSM environments, where the variability space is defined in meta-models, and code generators are developed against the meta-models. In [1], the authors present the idea of having a Framework-Specific Modeling Language (FSML) with support for round-trip engineering, based on manually defining meta-models with embedded mappings to the framework elements. In contrast to these approaches, ours encodes the DSML in a framework layer. Domain engineers are not required to manipulate and understand meta-modeling nor code generation technologies, but instead, they have to be familiar with aspect-oriented programming. Since in the conventional approaches the meta-models and tranformations can be defined freely, our approach is naturally less flexible concerning this issue. There is a trade-off between flexibility and automation. However, the results obtained when validating our approach on a complex framework such as Eclipse RCP, makes us believe that this trade-off is benefitial. Concerning round-trip engineering, our approach does not intend to support it, but we opt instead for having a clear separation between generated and manually written code, where the generated code is not intended to be manipulated. We considered the expressiveness of ALFAMA to be on the creative construction category, since the variability space is represented in a meta-model, equivalent to those used in MetaEdit+ or Microsoft DSL Tools.

In [19], the authors argue that implementing DSLs using general-purpose aspect-oriented languages can be advantageous over program generation, with respect to composability, scalability, and understandability. Our approach is a combination of aspect-orientation and program generation. We propose domain engineers to develop the specialization modules using aspect-oriented programming — a powerful means for composition — in order that application engineers can generate applications using a DSML — a means with expressive power and ease of use. We also favor the use of general-purpose aspect-oriented languages, but we strongly believe that aspect-orientation and program generation can complement each other in DSL implementation.

9 Conclusion

In this paper we presented an approach for extending an object-oriented framework with a specialization layer based on aspect-oriented programming, enabling automatic support for DSM. We validated the approach by implementing the ALFAMA prototype tool and performing a case study on Eclipse RCP. Our approach can be applied to product-lines implemented as object-oriented frameworks. A DSM approach based on what we propose allows the DSM support to be automatically up-to-date with a product-line platform, at the cost of having the additional specialization layer. As future work, we plan to provide support for helping on the development of specialization modules (e.g. through a pattern language), since this is the main difficulty when applying our approach. After carrying out the research presented in this paper, we strongly believe that it is possible to implement DSM support within the framework implementation. Adopting such an approach is well motivated by the difficulty of developing and maintaining code generators, the iterative nature of framework-based development, the unavoidable framework evolution, and obviously, the benefits of building applications using a DSML.

References

[1] Michal Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In *MoDELS*, 2006.

[2] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual mixin layers: aspects and features in concert. In *ICSE '06: Proceedings of the 28th international conference on software engineering*, 2006.

[3] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, 2003.

[4] Jan Bosch. *Design and use of software architectures: adopting and evolving a product-line approach.* ACM Press/Addison-Wesley Publishing Co., 2000.

[5] J. Craig Cleaveland. Building application generators. *IEEE Software*, 5(4):25–33, 1988.

[6] Krzysztof Czarnecki. Overview of generative software development. In *UPP*, 2004.

[7] DSM Forum. Workshops on domain-specific modeling, 2001-2006. http://www.dsmforum.org/DSMworkshops.html, 2007.

[8] Eclipse Foundation. AspectJ programming language. http://www.eclipse.org/aspectj, 2007.

[9] Eclipse Foundation. Eclipse platform and projects. http://www.eclipse.org/projects, 2007.

[10] Eclipse Foundation. EMF – Eclipse Modeling Framework. http://www.eclipse.org/emf, 2007.

[11] Eclipse Foundation. GMF – Graphical Modeling Framework. http://www.eclipse.org/gmf, 2007.

[12] Jack Greenfield and Keith Short. *Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools.* John Wiley and Sons, 2005.

[13] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, 2001.

[14] Juha Hautamäki and Kai Koskimies. Finding and documenting the specialization interface of an application framework. *Software: Practice and Experience*, (Electronic version):DOI 10.1002/spe.728, 2005.

[15] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1:22–35, 1988.

[16] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, 1997.

[17] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications.* Addison-Wesley Professional, 2005.

[18] MetaCase. MetaEdit+ tool. http://www.metacase.com.

[19] Mira Mezini and Klaus Ostermann. A comparison of program generation with aspect-oriented programming. In *UPP*, 2004.

[20] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *ACM Conference on Foundations of Software Engineering (FSE-12)*, 2004.

[21] Simon Moser and Oscar Nierstrasz. The effect of object-oriented frameworks on developer productivity. *Computer*, 29:45–51, 1996.

[22] Alvaro Ortigosa, Marcelo Campo, and Roberto Moriyón. Towards agent-oriented assistance for framework instantiation. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2000.

[23] Pure Systems. pure::variants. http://www.pure-systems.com/, 2007.

[24] André L. Santos. Automatic support for model-driven specialization of object-oriented frameworks using ALFAMA (to appear). In *OOPSLA'07 Demonstrations Track*, 2007.

[25] André L. Santos, Antónia Lopes, and Kai Koskimies. Framework specialization aspects. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, 2007.

[26] SourceForge. JHotDraw framework. http://www.jhotdraw.org, 2006.